# 第4章 検証を考慮して仕様/設計文書を書く

## ── UMLを利用する際に押さえておきたいこと

伊藤昌夫

**ここではUMLによる仕様/設計文書の書きかたについて述べる．UMLを利用する場合，各図の間にはそれぞれ関係があり，その一貫性を維持しながら仕様を表現しなければならない．そのため，いくつかの配慮が必要になる．ここでは簡単なモデルを示しながら，UMLで仕様を記述する際の注意点を紹介する．また，リアルタイム・システムを表現するための拡張記述についても紹介する．（編集部）**

今回，ソフトウェア設計品質ということについていろいろと考えてみました．わかっているつもり，すなわち設計については十分に経験を積んでいるつもりだったのですが，あらためて「設計品質」ということばを抜き出すと，説明がとても難しいことに気が付きました．機械の場合と違って，「設計結果図面ができたら，あとは製造の話」とはいきません．かなりの点で設計と（製造に相当する？）プログラミングが密接に結びついているからです．機械設計だと図面どおりに作るのが製造ですが，ソフトウェアの場合，そこまで図面を書こうとすると，プログラミングを行っているのとあまり変わらなくなってしまいます．したがって，「いかに抽象化するか」というのがソフトウェアの場合は大きな課題になります．

いかに抽象化するかという問題は，どのように仕様を書き，設計するかという問題でもあります．そこにはどうしても分析・設計手法が関係してきます．ここでは「オブジェクト間の協調を中心とする分析・設計」を考えて，その中で最終的に設計品質をどうとらえればよいかを説明したいと思います．道具としては，UMLの図式とその拡張を使うことにします．

### ● 仕様を決めることが難しい二つの理由

ソフトウェア設計において，仕様をどう記述するか，設計をどのように記述するかはつねに問題となってきました．なぜかというと，多くの場合，システムは人間のために存在するからです．人間が直接ハードウェアを操作する時代，それは単純なインターフェースしか持たず，人間はその数少ないインターフェースのみを利用していました．すなわち，人間がハードウェアに合わせていたと言えます（**図1**）．

現在では，よほどの単純なシステム以外は，例えばキーボードのようなインターフェースからなる入力機器を利用したり，ディスプレイの画面などを通じて複雑な出力を行うことになります．これにより，柔軟なユーザ・インターフェースを実現しています．

近年，ユーザの使いがってを改善するため，あるいはハードウェア自身が実現していない機能を受け持つため，ソフトウェアの規模が非常に大きくなっています．このことを仕様という側面で見たとき，以下の二つのことが言えます．

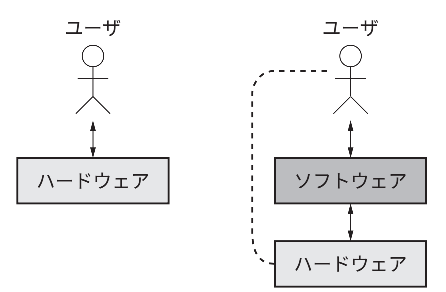



**図1　人間とハードウェアの間へのソフトウェアの介在を示す模式図**
ソフトウェアが媒介しないとき，（a）のようにユーザは基本的にハードウェアの提供する少数のインターフェースのみを使っていた．現在では，間にソフトウェアが介在することで，（b）のようにより柔軟なインターフェースおよびハード・ワイヤ化されないロジック（論理機能）の実現が可能となっている．図の矢印は主たるやりとりを示している．

● 人間系については容易に仕様が定まらない
● 機能が複雑化・大規模化しており，仕様として表現することが難しくなっている

## ● 仕様，設計，検証の関係

SLCP（Software Life Cycle Process）によれば，ソフトウェア開発の上流工程では，次のような作業が行われることになります．

● **ソフトウェア要求分析**
● **ソフトウェア・アーキテクチャ設計**
● **ソフトウェア詳細設計**

組織によって，あるいはプロジェクトによって，活動はより詳細化されますが，ここではおおむねこの分けかたを採用します．

今，ソフトウェア要求分析の結果としてソフトウェア要求仕様書（Software Requirement Specification，略してSRSと呼ぶこともある）があるものとします．また，ソフトウェア・アーキテクチャ設計の結果としてソフトウェア概要設計書が，ソフトウェア詳細設計の結果としてソフトウェア詳細設計書が作られるものとします．このとき，それぞれの文書の間には**図2**のような関係があると考えられます．この詳細化が正しいかどうかを調べる行為が「検証」と呼ばれています．例えば，ソフトウェア要求仕様書に書かれていることが，誤りなく，過不足なく，かつ一貫性を維持しながらソフトウェア概要設計書に記述されているかどうかを検査することになります．したがって，次のように言い換えることも可能です．「詳細化は開発として前進する方向で行われる行為である．それに対して，検証は逆向きに行われる行為である」．

ここまではわかりやすい話なのですが，実際にはそれほど明確に仕様と設計を分割することはできません．ソフトウェア要求仕様書を考えるとき，ある程度設計のことも考えなくてはなりません．そうしないと，実現しようのない要求仕様になるかもしれないからです．逆に，あらゆることを規定しようとすると，プログラム仕様書のようになってしまいます（例えば，エラー・メッセージの文面を決めるなど）．

ここでは，いくつかのキーワードに関してSyntropy手法[2]に従って定義しておきます．

● **「分析」とは，ソフトウェア開発プロジェクトにとって選択の余地のない（すなわち実現することを明示している）側面を見つけ，記述することである**
● **「設計」とは，ソフトウェア開発プロジェクトにとって選択可能な（すなわちまだ何も述べていない）側面を作り出し，記述することである**
● **「仕様」とは，ソフトウェア・システムの観測可能なふるまいの抽象的な記述である**

分析の結果が仕様であるとすると，この分析の定義にあるように，あくまでも（観測可能なシステムについて）実現すると宣言しているものが仕様の候補となります．したがって，現実的には，顧客によって仕様のレベルというのは随分と異なることになります．ただし，ここでは上記の“抽象的”というキーワードを強調しておきたいと思います．

## ● 良いソフトウェア要求仕様書とは？

仕様については，もう一つ参考になる記述があります．IEEE標準であるソフトウェア要求仕様に関する推奨です[3]．ここには，良いSRSの特性について次のような定義があり，重要な要素が明確に述べられています．

● **正確である**――正確であるとは，そこで述べられている要求が実現するべきもの（通常は，“システム”要求仕様書のような上位文書）と適合していること．
● **明白である**――ソフトウェア要求仕様書が明白であるというのは，そこで述べられている要求がたった一つの解釈しか持たないこと（用語が複数の意味を持つ場合は，用語集を通して意味を限定する必要がある．あるいは形式的な記述を用いてユニークな意味を与える）．
● **完全である**――完全であるとは次の三つの項目が成立するときである．第1に，すべての重要な要求（機能性や性能，設計制約，属性，外部インターフェース）が記述されていること．なお，システム要求仕様で定義されて

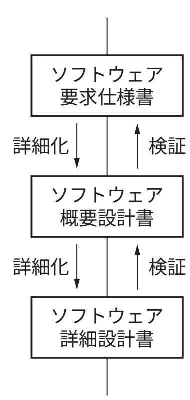



**図2**
**詳細化の過程**
**下位の文書は上位の文書を「詳細化する」という言いかたをする．例えば，ATMで現金を引き出すという要求仕様書上の機能に対して，概要設計書を読むことによって，どのような構成部品の関係でそれが実現可能かを知ることができる．あるいは，キャッシュ・カードを入出力する部分や，パスワードを確認する部分といった構成部品がわかる．詳細設計書を読むと，それぞれの構成部品の内部がどのように作られているかがわかる．**

いる外部の要求も，承認され，要求として扱われていること．第2に，すべてのあり得る状況における，すべての実現可能な入力データに対するソフトウェアの応答を定義すること．なお，無効な入力値への応答についても記述することは重要である．第3に，ソフトウェア要求仕様書の中のすべての図，テーブル，ダイヤグラムにラベルが付き，参照できること．なお，すべての用語と測定単位を定義していること．

- **一貫している**──上位文書と不整合がないこと．
- **重要性や安定性に従ってランク付けされている**──安定性については，予期される修正の数によって示すこともできる．
- **検証可能である**──ソフトウェア要求仕様書が検証可能であるというのは，ソフトウェア要求仕様書中のすべての要求が検証可能であるときのみである．
- **修正可能である**──ソフトウェア要求仕様書が修正可能であるとは，要求を変更しようとしたときに，ソフトウェア要求仕様書の構造とスタイルを維持したまま，容易に，完全に，一貫性を維持しながら変更できることを示す．
- **追跡可能である**──ソフトウェア要求仕様書が追跡可能であるとは，要求の起源が明確であり，以降の開発や文書の改訂において要求への参照が可能であることを示す．

先に，「実現すると宣言したものは仕様の一部」と定義しました．すなわち，設計の内容であってもそれがその宣言の対象となるならば，仕様に記述されることになります．IEEEのこの文書では，仕様に埋め込まれる設計内容として次の例が挙げられています．

「特別な場合では，いくつかの要求が厳しくデザインを制限するかもしれない．例えば，セキュリティや安全要求事項は，次のように直接設計について言及する場合がある．

a) いくつかの機能に関しては，別のモジュールとすること

b) プログラムのいくつかの領域では，制限された通信を行うこと

c) 重要な変数については，データの完全性を検査すること

設計制約に関する例は，物理的な要求/性能要求とソフトウェア開発標準/ソフトウェア品質保証基準である」

## ● モデルを記述するための道具“UML”

UMLとは，ご存じのようにプログラムを作成するときのさまざまな場面でアイデアを整理し，表現するための図式です．別のことばで表現するならば，プログラムを作るという行為は，特定のモデルを作る作業にほかならず，UMLはそのモデルを記述するための道具ということになります．

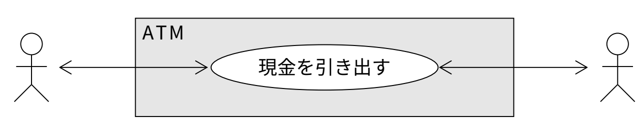
今，検証に焦点を当てます．そうしたときに，どうしたら検証がスムーズに行えるかを考えます．いちばんわかりやすい手段は，それぞれの段階でモデルの記述を形式的に（定型に）すればよいことになります．そうしておかないと，検証は，単に人間の注意深さに依存してしまいます．

ここで，いくつかのポイントがあります．以下では，機能の問題とふるまいの問題に分けて考えます．

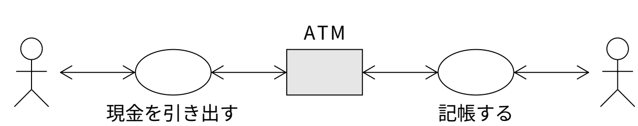
## ● UMLで機能を記述するポイント

仕様は，大きくは機能の記述と非機能の記述に分かれます（ここで非機能要件というのは，速度や使いがってに関わる要件）．ソフトウェアの場合，機能を見つけ出して記述するのはそれほど簡単ではありません．したがって，手法としていろいろなやりかたが提案されていますが，ここ

4




(**a**) 要求分析で使用する通常のユースケース




(**b**) ユースケース図形要素を用いた別のシステム要求に関する表現

**図3　要求分析で使用するユースケース図**

もちろん「記帳する」や「預け入れを行う」といったユースケースもATMにはあるが，ここでは例として一つだけ記述している（a）．「異なるアクタに対して，ユースケースは異なるイベント列を持つ」というのがユースケースの定義である．必要ならば，ユースケースは，それぞれイベント列を持つアクションへと詳細化される．このアクションに対して，後述する型仕様を記述する．これは，単なる記法の変更ではない．通常，異なるアクタは，同一ユースケースに対しても異なるイベント列を期待する．例えば，「口座のステータスを管理する」というユースケースは，アクタ「管理者」とアクタ「窓口」では異なるだろう．アクタ「管理者」の場合，入力はないかもしれないが，管理している全口座の情報を知る必要がある．一方，アクタ「窓口」は入力業務を行うが，知る必要のある情報は限定されている．オペレータの入力によって管理者はステータス管理を行うのだが，イベント列としては分けたほうが良い．これも重複オブジェクトの議論に似ていて，そのユースケースは別の側面をアクタに対して見せることになる．そのため，まずは違うものと考えるほうがよい．最初は(a)のように異なる側面をあまり意識することなく，一つで考える．しかし，きちんとシステムを仕様化したい場合に，不つごうな局面が多い．そこで仕様化直前では，(b)のような表記法が望ましい．

では筆者自身が用いている方法を示しておきます．もちろんまったく新しい方法というわけではなく，伝統的なSyntropyやCatalysisの手法に基づいています．

銀行のATM（現金自動預け払い機）を例にして考えます．最初に，ソフトウェア開発でよく利用されているユースケース図を用います．通常は**図3（a）**のように記述します．

ここでは，別の書きかたを考えてみます．理由の詳細な説明は省略しますが，**図3（a）**のような記述の場合，誤解されて利用されることが多いように感じるからです．システムの中にユースケースを書くと，ユースケース自身が機能を表現しているように見えます．ユースケースは，もちろんシステムが持つべき機能と無関係ではありませんが，機能そのものではありません．定義上は，あくまでもユーザ（アクタ）との間のトランザクション列に過ぎないのです．

**図3（b）**では，より端的に，アクタとシステムのやりとりとしてユースケースを定めています．さて，「現金を引き出す」はユースケースなのですが，システムの機能として考えた場合，「認証する」，「口座の状況から引き出し可否を確定する」，「実際に現金を出す」という三つの部分に分けることができそうです．今，それぞれを機能と呼んでもよいのですが，これ以上の分割が好ましくない最小単位として，「アクション」と呼ぶことにします．このアクション定義を正確に行うことで，次の仕様記述を行うことができます．

### ● オブジェクト中心かアクション中心か

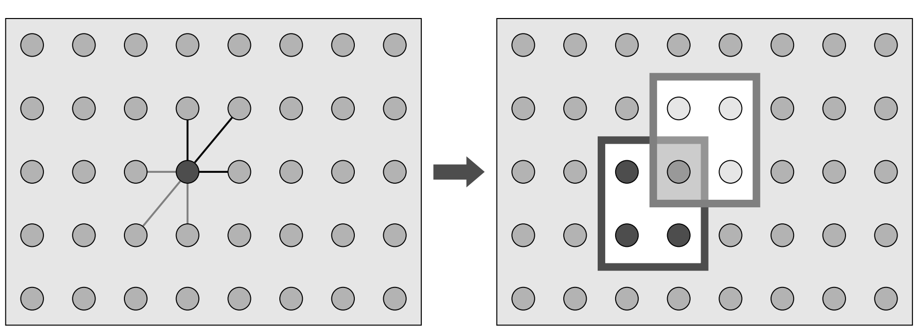
一般に，仕様の記述単位は機能ごと（本例ではアクションごと）になります．さて，先に見たソフトウェア要求仕様の特性（正確であること，明白であること，完全であること，一貫していることなど）をどのように実現するかが，つねに問題になります．ここでは「仕様型」という記述を通じて，この特性を満足することを考えます．

一つのアクションを記述するときに，もちろん文章で書いてもよいのですが，検証のため，中に出てくる語彙（ごい）を表現するのに「オブジェクト」を使用することにします．オブジェクト指向で言うオブジェクトです．「口座から現金を引き出す」というときに，このように日本語で書いてもよいのですが，もう少しだけ正確な表現を目指すことにします．

オブジェクトは，ほかと協調して何か役に立つこと（それが，まさに機能になる）をするものです．ですから，個別のオブジェクトを意識するのではなく，オブジェクトどうしの協調を意識する必要があります．私にとっての関心事は，口座オブジェクトがあることではなく，出金というアクションに対して，例えばログ・オブジェクトに記録が残り，ATMオブジェクトに対して通知がいくということなのです．

もちろん，それでもオブジェクトが登場するわけですから，どこか鶏と卵のような関係があるのですが，アクションをきちんと説明できるか否かをオブジェクト識別の判断に使います．オブジェクト識別が主ではなくて，アクションを説明するためにはどういうオブジェクトがあればよいかを考えることになります．

システム内で中心となるオブジェクトは，複数のアクションによって何度も使われます．例えば，ATMシステムにおいて顧客オブジェクトは，出金においても入金においても登場することになります．重複するオブジェクトは，多くの場合，各アクションで異なる役割を果たしています．例えば，口座オブジェクトは自分が管理している金額（残高）の減少，あるいは増加について責任を持ちます．その一方で，残高照会（というアクション）においては，ATM



（a）問題空間に散在するオブジェクト群　　（b）アクションと関連するオブジェクト

**図4　問題空間のオブジェクト**

丸い部分がシステムを構成するオブジェクトを表す．（b）の四角で囲まれた部分は，あるアクションを構成するオブジェクトの集合を表す．

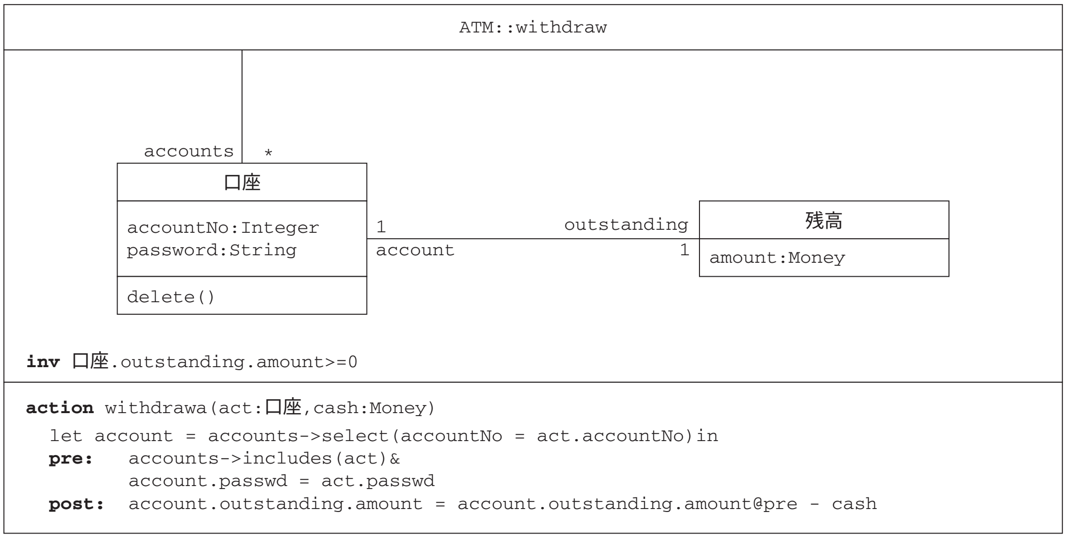



**図5　型仕様の例**
**ATMで口座から現金を引き出すときの型仕様を示す．**

オブジェクトに対して残高を答えます．このように各オブジェクトはアクションごとに異なる側面（役割）を持つということになります．

こういった役割の重複は，最終的に調整して整合性を確保しなければなりません．口座オブジェクトというのは，一つのオブジェクトに過ぎないわけですから．最終的に行うこの整合性の確保は，多少めんどうにも感じられます．しかし，逆に考えるなら，多くの役割の重複を持つオブジェクトは，システムにおいて核となるオブジェクトです．本来的に複数の側面を持ち，もともとさまざまな視点を通して考えるべき対象です．最初に何もないところから，全体とその全体を構成するオブジェクトの役割を考えるより，こうしてアクション単位で考え，後で整合性を取るほうが，（とくに規模の大きなシステムにおいては）考えやすいだろうと，経験上からも思っています．

**図4（a）**は，問題空間上のオブジェクトを表しています．一般的な解法の場合，この広い空間上の多数のオブジェクトを見つけだして，その間の静的な関係を見つけだすところから作業が始まります．これはモノとしてのオブジェクトを見つける，という説明に通じるかもしれません．しかし，これはそれほど容易なことではありません．

これらのことより，アクションを中心とし，関連するオブジェクトを登場させることで，適切に仕様を記述できるということがわかると思います．次に，実際の仕様の書きかた（**図4（b）**の四角で囲んだ部分をどのように表現するか）を説明します．

4

### ● 型仕様から全体の仕様を構成する

型仕様は，各アクションの形式的な定義です．仕様は基本的にこの型仕様から構成されます（個別の型は仕様型と呼ぶ）．例を挙げます．**図5**は，ATMで口座から現金を引き出すときの型仕様による例になります．

型仕様は，基本的にクラスと同じ構造を持ちます（しかし，いわゆるプログラム的なクラス図ではないことに注意）．最上段ではアクション名を記述します．アクション名は，「型名::アクション名」という形式で表現します．ここではATMという抽象データ型のアクション「現金を引き出す（withdraw）」を定義することにします．

型仕様の図の中段は，ここではATM型の静的な側面を表します．通常のクラス表現では属性を記述する部分です．属性の代わりにユーザ型を図式表現していると思っていただいても大きな違いはありません．

下段は，アクション定義の部分です．クラスの場合は操作を記述する部分です．このアクション定義部分では，UMLの仕様中で定義されているOCL（Object Constraint Language：オブジェクト制約言語[5]）を用いて記述します．

この下段のコンパートメントで定められるのは表明（assertion）であり，型が取りうる状態を明示的に表現した

い場合は，その型についてのステート・チャートを作成する必要があります．それはほかの手法と同じです注1．ちなみに，表明は次の要素からなります．

- **不変条件（invariant）** ── アクションを通じて変わらない条件．あるいは，中段で現れる内部クラス間の関係のうち，図式で表現するのが難しい条件．
- **事前条件（pre-condition）** ── そのアクションを実行するにあたって成立していることが必要な条件（状態）
- **事後条件（post-condition）** ── そのアクション実行後に成立している条件（状態）．すなわち，アクションのゴール．

見かけ上はクラスがクラスを包含した形になっています．**図4（b）**と比較してみてください．この図の中の小さな丸が内部型に相当し（**図5**の口座クラスや残高クラス），右側の四角で囲まれた領域が仕様型になります．この包含関係がある分だけ抽象度が上がっていることになります．

この型仕様は，初期段階では分割して複数人で作成することが可能です．もちろん，先に述べた重複オブジェクトの問題があるので，ある段階からは内部に含まれる型に関する整理・統合が必要になります．この整理・統合により，ふるまい記述部（アクションの定義部）も影響を受けます．ただし，すべてではありません．あくまでも型仕様においては，必要な情報のみを記載すればよいのです．そうすることで，重複部は最小で済みます．

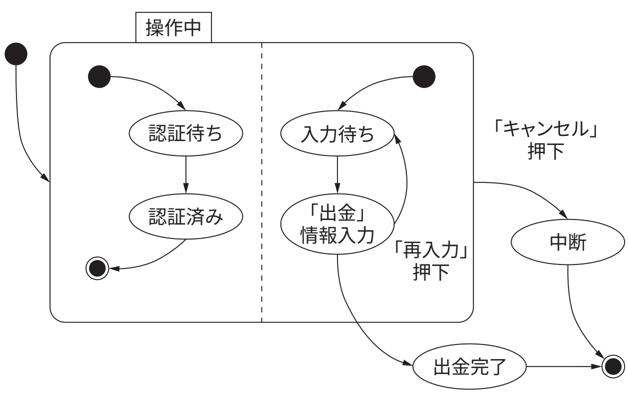
大規模アプリケーションにおいて仕様モデルを単にクラス図で表現しようとすると，とても印刷できないほど膨大なクラス図を相手にすることになり，理解することすら困難です．ここで紹介した手法では，必要なクラスのみを出現させますから，アクション（機能）の理解はずいぶんと簡単になります．その一方で，同一のオブジェクトが複数のアクションの中で現れることにもなるので，一貫性を維持することは困難になります．何らかのツールの手助けを得るのが望ましいと思います注2．

### ● 複雑な状態が含まれるときはステート・チャート

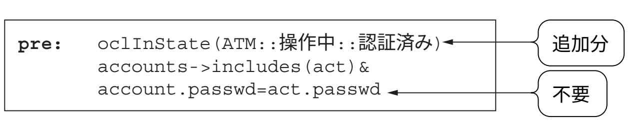
先の機能表現だけでも十分なのですが，複雑な状態を表現することが必要な場合もあります．これは，UMLでは有限状態機械モデル（ステート・チャート）で表現することになります．ふるまいは，今いる状態に依存して異なります．

例を考えてみます．今，ATMで主として現金を下ろすということについて，**図6（a）**のようなステート・チャートを書いてみます．ユーザ認証が最初にかならず行われない場合もあるので，ここでは並行表現を使っています注3．アンド・ラインと呼ばれる点線でしきられた二つの状態遷移は，それぞれ独立に動作できます．ただし，明らかに振り込みが完了するためには，ユーザ認証が終わっていなければなりません．この場合，先ほどの仕様型の中の記述に**図6（b）**を追加します．

以前の事後条件では，「入力パスワードと口座の属性であるパスワードが等しいこと」という条件にしていました．ステート・チャートを併用する場合は，その代わりに，「“認証済み”という操作中のサブステートにいること」という条件に変えています．




（**a**）出金にかかわるATMの状態遷移図




（**b**）状態指示の追加

**図6　状態遷移図の例**
**ATMにおいて，主として現金を下ろすということに関する状態遷移図を示す．**

注1：集合を直接扱うこと，これはOCLが得意とするところである．例えば，新規に口座を開くというのは，その支店の口座数が1増えることとして表現できる．一方，すべての述語（説明）が集合に還元できるわけではない．オブジェクトのある状態を表現する場合はOCLで有効に記述することは難しく，一般のプログラムと同じように属性を用いて表現したり，状態遷移図（ステート・チャート）と組み合わせて表現することになる．

注2：OCLは仕様記述言語から多くのアイデアを得ているので，各種のツール（タイプ・チェッカやスタイル・チェッカ）の恩恵を受けられそうだ．ただし，あくまでも図式と同時に用いることを想定しているので，OCL部分だけの検査にはあまり意味がない．図式と合わせて検査するという試みも大学などで行われているが，まだ実用の域には達していない．

注3：それは多分に，認証行為だけでトランザクションを発行すると処理時間がかかるので，最後にまとめて行うのだろう．ただし，これは設計判断だという言いかたもできるが，ここでは説明のために，仕様上この有限状態機械モデルを使うことにする．

基本的には同じとも言えるのですが，例えば「3回まちがえるとしばらく出金操作ができない」という仕様を書きたいとき，ステート・チャートを使わないと少しめんどうな記述が必要になります．OCLは，アクションの途中経過を気にせず，アクションの開始条件と終了条件のみを記述するためです．その割り切りのおかげで効率の良い記述が可能となるのですが，3回というのを詳細化せずに書こうとするとたいへんです．

#### ● UMLを使って検証するときの視点

UMLの個々の図を見たとき，単純な確認項目をいくつも挙げることができます．例えば，「名称が妥当か」，「クラス図において，不用意に循環的な関連を引いていないか」，「状態機械図において，遷移できない状態はないか，脱出できない状態はないか」といったことです．

UMLでは13種類の図式がありますが，それぞれに関係があります(**図7**)．例えば，先に見たように，仕様型の中に"認証済み"状態があることを事前条件で宣言しているので，仕様型図はステート・チャートに依存していることになります．一方，ステート・チャートは仕様型ごとに記述することになるので，逆にステート・チャートは仕様型図に依存していることになります．したがって，この両者の間では一貫性が確保されている必要があります．

仕様は詳細化されて設計になります．したがって，設計は仕様に適合していなくてはなりません．仕様型では，先に述べたように表明を用いて記述を行います．設計において，この表明は次のように維持されている必要があります．

- **事前条件が厳しくなってはならない**
- **事後条件が緩くなってはならない**
- **不変条件は維持されなくてはならない**

事前条件は，そのアクションが実行されるための前提となります．これが厳しくなると，仕様上で問題なかった前提が実際には満足されなくなります．前提を満足しない場合は気にしないという立場をとるので，前提が緩んでも事後を満足するということならば，問題がないことになります．

事後条件は，アクションの達成目標を示します．したがって，それが緩和するということは目標に達しないわけですから，適合しないことになります．例えば，先の出金の例だと，「現金が出てきて，出金分だけ口座残高が減少する」というのが事後条件でした．もし，非機能要件に「使いがってが良いこと」という制約があり，現金に1万円が含まれる場合，両替できるように設計したとします．これはもともとの事後条件を満足したうえで新たな制約を加えているので，問題ないということになります．

不変条件については，一般に厳しくすることも緩和することもありません．

ある仕様から，その仕様に適合する複数の設計を考えることができます．「なぜそのような設計になったのか」という設計意図の記述はつねに必要です．その設計意図がないかぎり，適合しているというこれまでの評価基準だけでその設計が良いか否かを判定することは，非常に困難です．

#### ● リアルタイム・システムの仕様書を書く

前述のように，ソフトウェア要求仕様書および設計書の記述法にはさまざまな書きかたがあります．ここでは，仕様型および設計型に基づく記述例を示します．



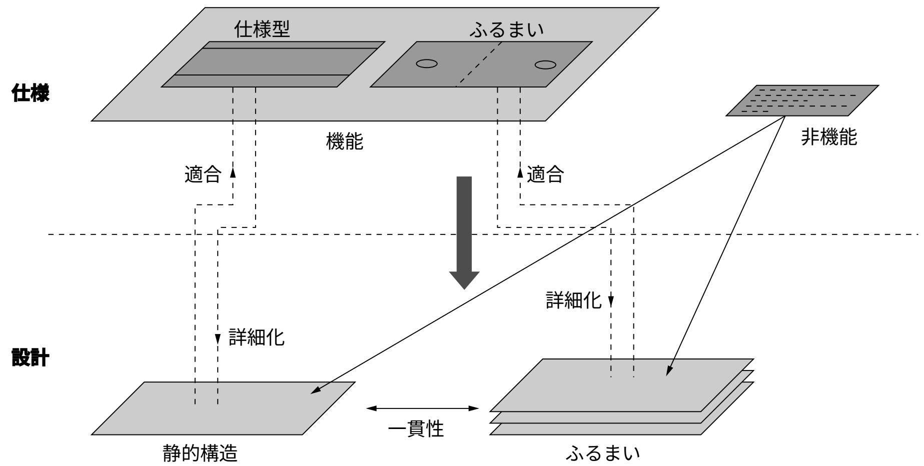



**図7**
**仕様と設計モデルの間の関係**
UMLでは各図のそれぞれに関係がある．このことから，単純な確認項目をいくつも挙げることができる．

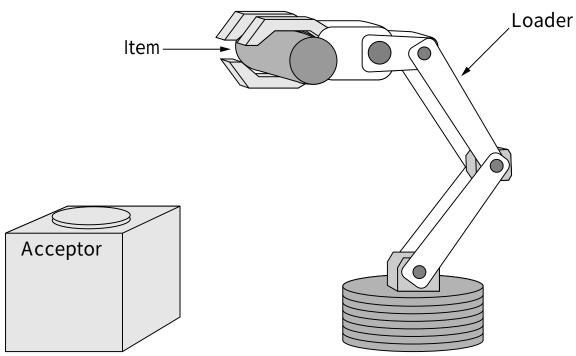



**図8　リアルタイム・システムの例**
単純なアーム（Loader）があり，何かの物体（Item）を運ぶ．受け手側（Acceptor）は準備状況に応じて，物体を受け取る．

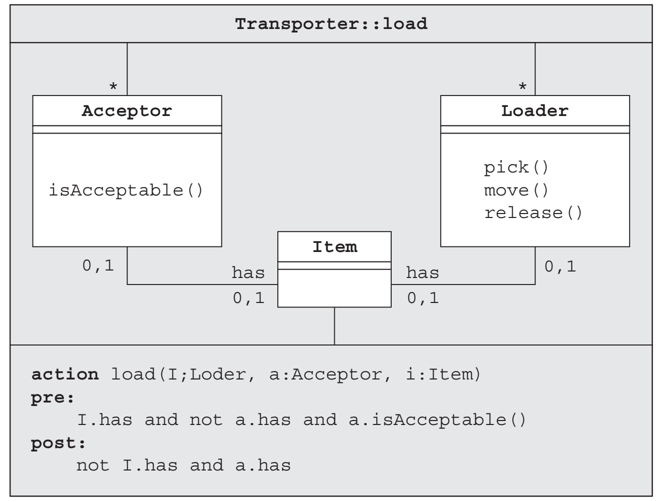



**図9　仕様型Transporterのアクションloadの仕様記述**
ソフトウェア要求仕様書の仕様型に基づく記述例を示す．

例として，**図8**のような状況を想定します．要件は以下のとおりです．

【要件】
単純なアーム（Loader）があり，何かの物体（Item）を運ぶ．受け手側（Acceptor）は準備状況に応じて，物体を受け取る．

**1） 仕様記述**

先に述べた仕様型に基づく結果を**図9**に示します．

事前条件は次のように考えます．

- まず，そもそもアイテムが存在しなくてはならない（もちろん，テストのための空動作があるかもしれないが，それは`load`というアクションではないとしている）
- ここでは，`Acceptor`側に重ね置きはできないという前提にしている
- かつ，何らかの受け入れのための準備動作が必要だとしている（`isAcceptable()`）

事後条件は次のようになります．

- `Loader`はすでにアイテムの移動を完了しているわけなので，アイテムは持っていない
- 逆に`Acceptor`は，`Item`を持っている必要がある

**2） 時間記述**

UMLを用いて時間記述を明示的に行うのは，それほど簡単ではありません．それはUML 2.0になっても状況は変わっていません．あえて書くとすると，シーケンス図に注記として入れるか，タイミング・チャートを使用するかです．しかし，これらの方法では，仕様型をはじめとするクラス図やステート・チャートと一貫性を確保することが難しくなります．したがって，何らかの方法で拡張しなければなりません．

一つの拡張は，十数年前にDavid Harel氏がステート・チャートを考案したときに行われた拡張です(6)．UMLでは，状態遷移の上に次のような形で遷移条件などを記述することができます．

```
[ガード条件]　イベント名　/　アクション名
```

例えば，「ドアを閉めろ」という指示が来たとき，じゃまなものがなければドアを閉めてかぎをかける，というのは，次のように書けます．

```
[doorWay->isEmpty] Shut/ CloseAndLock
```

ところで，ここで，「ドアを閉めてかぎを掛けろ」アクションは，ガード条件が成立したうえで，Shut指示が着いてからある程度の時間をおいて実行しなければならないとします．この“時間をおいて”というのが通常の記述では書けません．基本的にトリガは一瞬で伝わりますし，アクションは一瞬で実行されるという前提に立っているからです．

ところが，リアルタイム処理では有限時間の扱いが重要になります．そのためにいくつかのキーワードを用います．例えば，`tm(E, N)`というキーワードがあり，これは「`N`クロック後にイベント`E`を発行する」という意味になります．したがって，「10クロック後に`CloseAndLock`を発行する」というのは，このキーワードを使って次のように書けます．

```
[doorWay->isEmpty] Shut/tm(CloseAnd
Lock, 10)
```

もちろん，このアクション部分の定義では10クロック後にイベントを発行するだけになるので，実際にこのイベントを受けて処理を行う部分を別途定義しなくてはなりません．

こういったステート・チャートの拡張はその後も行われていますが，やはりこのままだと状態遷移（ステート・チャート）と仕様型（クラス図）の間で一貫性を保つのが難しくなります．ここでは，仕様型をそのまま使って，時間記述を行う方法を考えます(7)．

今，「Loderは40～80クロック内で積み込みを完了しなくてはならない」ということを時間制約として記述したいとします．積み込み完了状態を，"loaded"状態と名づけると，次のように書くことができます．@post[40, 80]というのが拡張記述の部分です．

```
l@post[40, 80] -> includes(l::loaded)
```

oclInStateを使わずincludesを使っているのは，状態遷移そのものも，状態の集合とその並びというように，集合の枠組みの中で定義したいからです．これによって，仕様型が集合に基づいているように，状態も含めて一貫性を持って扱えることになります．同じ枠組みで扱えるメリットは，ツールによる検証やシミュレーションを行えるようになることです．

図式とテキストの両方をツールで扱うというのは，もともとたいへんなことです．できれば，図式は少なく，できるだけテキストで表現したほうが機械は扱いやすいのです．もちろん，人間はテキストばかりでは辛くなるので図を使いたいわけですが，ここで紹介した時間記述はその折衷をねらっています．

## ● 紹介した手法は確実に実践可能

ソフトウェアの仕様や設計の検証ということでは，さまざまな研究や実践がなされています．今回紹介したのは，どの場面でも確実に実践可能な方法です．冒頭で述べたように，ソフトウェアの場合は検証の対象となる仕様書そのものの記述が難しいという事実があります．ここがあいまいにしか書けないと，何も検証するべきことがなくなってしまいます．ですから，今回紹介した方法は，あまたある方法のうち，検証可能性を持った方法とも言えます．

また，途中でツールに関して何度か触れました．回路設計と比べると利用の頻度は随分と低いと思います．ただ，それはソフトウェア・エンジニアの「紺屋の白ばかま」というわけではなく，最初に述べたようにソフトウェアの仕様記述には難しさがあるからでしょう（あるいは，そう思いたい）．それはきっと抽象化の難しさだろうと筆者は思っています．

### 参考・引用*文献


(1) Information technology - Software life cycle processes, ISO/IEC 12207, 1995.
(2) Cook, S., Daniels, J.; *DESIGNING OBJECT SYSTEMS*, Prentice Hall, 1994注4.
(3)* IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1998, 1998.
(4) D'Souza, D., et al.; *OBJECTS, COMPONENTS, AND FRAME WORKS WITH UML - THE CATALYSIS APPROACH -*, Addison-Wesley, 1999注5.
(5) OCL 2.0 Specification Version 2.0, OMG ptc/2005-06-06, 2005注6.
(6) Harel, D., Politi, M.; *MODELING REACTIVE SYSTEMS WITH STATECHARTS*, McGrawHill, 1998注7.
(7) Flak, S.; *UML-BASED SPECIFICATION OF STATE-ORIENTED REAL-TIME PROPERTIES*, Shaker Verlag, 2003.



**いとう・まさお**
**(株)ニルソフトウェア**


**<筆者プロフィール>**
**伊藤昌夫．自動車会社，航空機関連会社のソフトウェア・エンジニアを経て，ニルソフトウェアを設立．ソフトウェア・プロセスおよび開発環境が専門で，そのためのコンサルテーションおよびツールの提供を行っている．設計における人間の認知活動に興味を持っている．**

注4：Syntropyについて書かれた唯一の書籍．この書籍で記述された内容が，今のOCLのベースになっていると言われている．ただし，クラス図をベースとする仕様型とは違って，Syntropyでは，ステート・チャートとOCL風の記述を組み合わせている．リアルタイム系の記述に重点をおいていたためだろう．
注5：今回の仕様型に関する基本的な内容はここに書かれている．Syntropyの影響を強く受けている．
注6：1.xと比べるとずいぶん使いやすくなった．ただし，2.0は2005年11月末現在，正式のOMG仕様にはなっていない．
注7：発行年は1998年だが，その前からSTATEMATEというツールに実装されて使用されていた．書籍という形で記述されたのがこの年になる．233ページ以降にアクション表現がまとめられている．同じころ，米国Carnegie Mellon UniversityのSoftware Engineering Institute（SEI）でSTATEMATEを使ったソフトウェア開発プロセスの記述とシミュレーションの実験が行われた．